On the Scalability of Proof Carrying Code for Software Certification*

Andrew Ireland

School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, Scotland, UK
a.ireland@hw.ac.uk

Abstract. Proof Carrying Code provides an approach to software certification, where trust management is decentralized. The approach has been successfully applied to relatively simple properties. Here we consider the scalability of the approach when more comprehensive properties are considered, *e.g.* functional properties. We argue that tactic-based theorem proving, and in particular proof plans, have a role to play in addressing the issue of scalability.

1 Introduction

Within the *Proof Carrying Code* (PCC) paradigm [7], software certificates correspond to formal proofs, *i.e.* a proof that a program satisfies a given safety policy. The responsibility for proof construction lies with the code producer, while a relatively light-weight proof checking process is all that is required on the consumer side. Moreover, the consumer does not need to trust the producer or any third-party intermediaries. As a consequence, PCC decentralizes trust management, *i.e.* the *trusted computing base* is minimal and local to the consumer.

Initially, proofs were relatively large, given the size of code involved. Significant progress, however, has been made in reducing the size of proofs, *i.e.* software certificates. In particular, an approach known as *Oracle-based Proof Carrying Code* (OPCC) uses *oracle strings* [8] as a means of representing the minimal information required for proof checking, *i.e.* the checker is only provided with information when a choice is required. Proof *tactics* have also been used to reduce the size of proofs, *i.e.* large proof steps defined in terms of tactics. This is known as *Tactic-based Proof Carrying Code* (TPCC) [1]. Of course within TPCC, the tactic definitions are required for proof checking, thus increasing the machinery on the consumer side.

PCC has been mainly concerned with safety properties, such as type safety and memory management safety. The relative light-weight nature of these properties has meant that proof construction corresponds to type inference. The need for more comprehensive properties is widely recognized. For instance, the MOBIUS project[1] has identified the need for comprehensive policies, such as functional properties, as one of the *"challenges that lie far beyond the current state-of-the-art"*. Meeting this challenge will increase the burden of proof associated with PCC, both in terms of proof construction and communication. Below we explore these issues in more detail.

2 Proof Construction

Extending PCC to include functional properties introduces all the complexities that are associated with software verification, *e.g.* the need for code to be annotated

* The work discussed was supported in part by EPSRC grant GR/S01771.

[1] MOBIUS: `http://mobius.inria.fr/twiki/bin/view/Mobius`.

with auxiliary assertions, such as loop invariants. The current focus on type-based methods will need to be combined with logic-based methods. In particular, theorem proving and program analyzers that assist with the generation of code annotations will be required.

We believe that the technique known as *proof planning* [3] also has a role to play here. Proof planning is a computer-based technique for automating the search for proofs. At the core of the technique are high-level proof outlines, known as *proof plans*. A proof plan embodies a *generic* tactic and is typically hierarchical in structure. Proof planning is the process by which a customized tactic is constructed for a given conjecture. The generic nature of a proof plan makes for a robust style of reasoning, *i.e.* proof planning can deal with changes to a conjecture, as long as the changes fall within the scope of the given proof plan. The use of proof planning to support proof construction would therefore represent a natural extension to TPCC. In terms of program analysis, proof planning has also demonstrated its value through the NuSPADE project[2], where proof planning was investigated within the context of verifying software written in SPARK [2]. In particular, proof-failure analysis, a key feature of proof planning, was used in conjunction with program analysis to guide the generation of loop invariants [4–6].

3 Proof Communication

As noted above, OPCC and TPCC have achieved significant reductions in the size of proofs. It is unclear, however, whether or not these approaches will scale to meet the challenges associated with more comprehensive properties. Here we propose an alternative approach. Instead of communicating a proof, or *how* to construct a proof (via a tactic or proof oracle), we propose communicating *what* knowledge is required in order for the consumer to re-construct a producer's proof. What we will refer to as *Proof Plan Carrying Code* (PPCC), can be viewed as an extension of TPCC. To achieve PPCC, we envisage the notion of a *Proof Planning Oracle* (PPO) , *i.e.* information on which proof plans and theories were used in planning a particular conjecture or class of conjectures. We see PPOs as an optional input/output to the existing proof planning framework. That is, the producer will use a proof planner to generate a PPO which is then used to constrain proof planning on the consumer side.

While PPOs will significantly reduce the size of software certificates, it will also significantly increase the burden on the code consumer. Firstly, the code consumer will require access to the proof plan and theory repositories referenced by the PPO – introducing the problem of managing distributed repositories. Secondly, the code consumer will be required to run a proof planner as well as a proof checker – increasing the consumer's computational overhead. Note however that the PPO will significantly reduce the search involved in re-constructing proofs on the consumer side. As is the case with TPCC, this additional overhead will exclude on-device proof checking. For many applications this would be a show-stopper, *e.g.* smart card applications with minimal resources. However, we believe that such applications will also rule-out on-device proof checking with respect to the more comprehensive properties that are currently being considered. So where on-device checking is not essential, but where comprehensive properties are mandatory, then PPCC may provide a practical approach.

[2] NuSPADE: `http://www.macs.hw.ac.uk/nuspade`.

4 Conclusion

PCC has been applied successfully to relatively simple properties. Targeting more comprehensive properties raises questions about the scalability of current approaches. We have argued that proof plans have a role to play in addressing the scalability of proof construction. In terms of representing software certificates, we have proposed the use of proof planning oracles as a technique for reducing the size of formal proofs.

References

1. D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile resource guarantees for smart devices. In *Proc. Intl. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2005.
2. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, 2003.
3. A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.
4. B.J. Ellis and A. Ireland. Automation for exception freedom proofs. In *Proceedings of the* 18^{th} *IEEE International Conference on Automated Software Engineering*, pages 343–346. IEEE Computer Society, 2003. Also available from the School of Mathematical and Computer Sciences, Heriot-Watt University, as Technical Report HW-MACS-TR-0010.
5. B.J. Ellis and A. Ireland. An integration of program analysis and automated theorem proving. In E.A. Boiten, J. Derrick, and G. Smith, editors, *Proceedings of 4th International Conference on Integrated Formal Methods (IFM-04)*, volume 2999 of *Lecture Notes in Computer Science*, pages 67–86. Springer Verlag, 2004. Also available from the School of Mathematical and Computer Sciences, Heriot-Watt University, as Technical Report HW-MACS-TR-0014.
6. A. Ireland, B.J. Ellis, A. Cook, R. Chapman, and J. Barnes. An integrated approach to program reasoning. Technical Report HW-MACS-TR-0027, School of Mathematical and Computer Sciences, Heriot-Watt University, 2004.
7. G. C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, jan 1997.
8. G.C. Necula and S.P. Rahul. Oracle-based checking of untrusted software. In *POPL: 28th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2001.